TCP

Dannis Lin
NCTU CIS Operating System lab.
2002 Linux kernel trace seminar

Socket family

PF_UNIX,PF_LOCAL	Local communication
PF_INET	IPv4 Internet protocols
PF_INET6	IPv6 Internet protocols
PF_IPX	IPX - Novell protocols
PF_NETLINK	Kernel user interface device
PF_X25	ITU-T X.25 / ISO-8208 protocol
PF_AX25	Amateur radio AX.25 protocol
PF_ATMPVC	Access to raw ATM PVCs
PF_APPLETALK	Appletalk
PF_PACKET	Low level packet interface

Socket type

- **SOCK_STREAM**

  Provides sequenced, reliable, two-way, connection-based byte streams. An out-of-band data transmission mechanism may be supported.

- **SOCK_DGRAM**

  Supports datagrams (connectionless, unreliable messages of a fixed maximum length).

- **SOCK_RAW**

  Provides raw network protocol access.

Socket protocol

- The protocol specifies a particular protocol to be used with the socket.
- Normally only a single protocol exists to support a particular socket type within a given protocol family.
- However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner.

sys_socketcall

/net/socket.c

```
asmlinkage long sys_socketcall(int call, unsigned long *args)
{       .........
        switch(call)
        {
           case SYS_SOCKET:
              err = sys_socket(a0,a1,a[2]);
              break;
           case SYS_BIND:
              err = sys_bind(a0,(struct sockaddr *)a1, a[2]);
              break;
           .........
           case SYS_SEND:
              err = sys_send(a0, (void *)a1, a[2], a[3]);
              break;
           default:
              err = -EINVAL;
              break;
        }
        return err;
}
```

Socket Creation

sys_socket

```
asmlinkage long sys_socket(int family, int type, int protocol)
{
        int retval;
        struct socket *sock;
        retval = sock_create(family, type, protocol, &sock);
        if (retval < 0)
            goto out;
        retval = sock_map_fd(sock);
        if (retval < 0)
            goto out_releas;
out:
        return retval;
out_release:
        sock_release(sock);
        return retval;
}
```

Return a file descriptor

sock_create

```
int sock_create(int family, int type, int protocol, struct socket **res)
{
        int i;
        struct socket *sock;
        ……
        if (!(sock = sock_alloc()))
        {
           printk(KERN_WARNING "socket: no more sockets\n");
           i = -ENFILE;
           goto out;
        }
        sock->type = type;
        if ((i = net_families[family]->create(sock, protocol)) < 0)
        {
           sock_release(sock);
           ………
        }
…
}
```

allocate a socket

static int inet_create(struct socket *sock, int protocol)

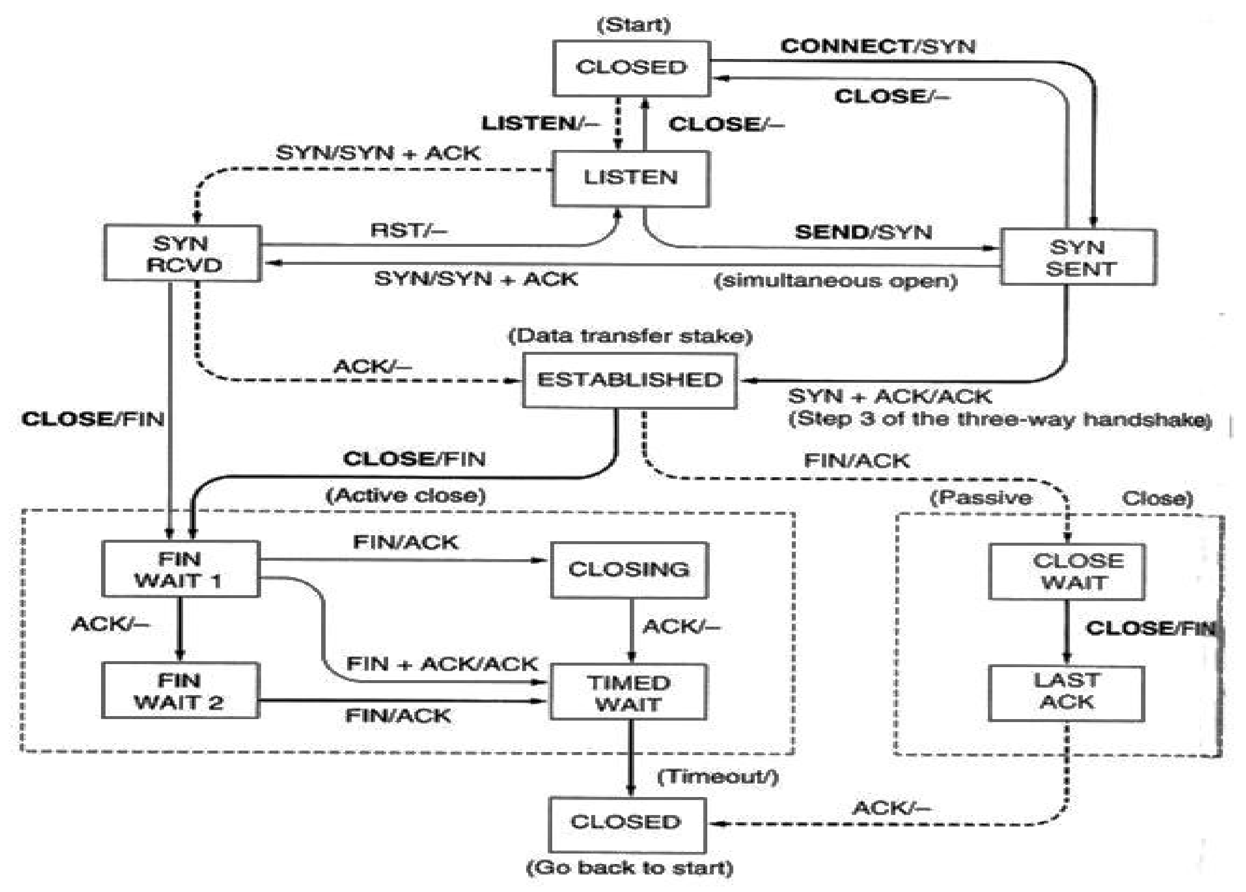
inet_create

```
static int inet_create(struct socket *sock, int protocol)
{
        struct sock *sk;
        ……..
        sock->state = SS_UNCONNECTED;
        sk = sk_alloc(PF_INET, GFP_KERNEL, 1);
        if (sk == NULL)
           goto do_oom;
        ……..
        sk->family = PF_INET;
        sk->protocol = protocol;
        ……..
        if (sk->prot->init) {
           int err = sk->prot->init(sk);
           if (err != 0) {
              inet_sock_release(sk);
              return err;}
        } return 0;
}
```

Allocate a sock

```
static int tcp_v4_init_sock(struct sock *sk)
{
    ….
    tcp_init_xmit_timers(sk);
    tcp_prequeue_init(tp);
    tp->rto = TCP_TIMEOUT_INIT;
    tp->mdev = TCP_TIMEOUT_INIT;
    ….
    sk->state = TCP_CLOSE;
    ….
    return 0;
}
```

TCP finite state machine



TCP state

```
enum
{
        TCP_ESTABLISHED = 1,
        TCP_SYN_SENT,
        TCP_SYN_RECV,
        TCP_FIN_WAIT1,
        TCP_FIN_WAIT2,
        TCP_TIME_WAIT,
        TCP_CLOSE,
        TCP_CLOSE_WAIT,
        TCP_LAST_ACK,
        TCP_LISTEN,
        TCP_CLOSING,
        TCP_MAX_STATES
};
```

Socket Listen

sys_listen

```
asmlinkage long sys_listen(int fd, int backlog)
{
        struct socket *sock;
        int err;
        if ((sock = sockfd_lookup(fd, &err)) != NULL)
        {
            if ((unsigned) backlog > SOMAXCONN)
                backlog = SOMAXCONN;
            err=sock->ops->listen(sock, backlog);
            sockfd_put(sock);
        }
        return err;
}
```

```
int inet_listen(struct socket *sock, int backlog)
```

```
extern __inline__ void sockfd_put(struct socket *sock)
{ fput(sock->file); }
```

inet_listen

```
int inet_listen(struct socket *sock, int backlog)
{
        struct sock *sk = sock->sk;
        unsigned char old_state;
        int err;
        lock_sock(sk);
        err = -EINVAL;
        if (sock->state != SS_UNCONNECTED || sock->type != SOCK_STREAM)
            goto out;
        old_state = sk->state;
        if (!((1<<old_state)&(TCPF_CLOSE|TCPF_LISTEN)))
            goto out;
        if (old_state != TCP_LISTEN) {
            err = tcp_listen_start(sk);
            if (err) goto out;
        }
        sk->max_ack_backlog = backlog;
        err = 0;
out:
        release_sock(sk);
        return err; }
```

Close state: start to listen
Listen state: remain listen

tcp_listen_start

```
int tcp_listen_start(struct sock *sk)
{
      ……
      sk->max_ack_backlog = 0;
      tcp_delack_init(tp);
      sk->state = TCP_LISTEN;
      if (sk->prot->get_port(sk, sk->num) == 0)
      {
         sk->sport = htons(sk->num);
         sk_dst_reset(sk);
         sk->prot->hash(sk);
         return 0;
      }
      ……
      sk->state = TCP_CLOSE;
      return -EADDRINUSE;
}
```

Call function : tcp_v4_get_port()

Obtain a reference to a local port for the given sock,if num is zero it means select any available local port.

Another socket is already listening on the same port.

Socket Connect

sys_connect

```
asmlinkage long sys_connect(int fd, struct sockaddr *uservaddr, int addrlen)
{
        struct socket *sock;
        char address[MAX_SOCK_ADDR];
        int err;
        sock = sockfd_lookup(fd, &err);
        if (!sock)
            goto out;
        err = move_addr_to_kernel(uservaddr, addrlen, address);
        if (err < 0)
            goto out_put;
        err = sock->ops->connect(sock, (struct sockaddr *) address, addrlen, sock->file->f_flags);
out_put:
        sockfd_put(sock);
out:
        return err;
}
```

The address is in user space so we verify it is OK and move it to kernel space.

Call function: inet_stream_connect()

inet_stream_connect (1)

```
int inet_stream_connect(struct socket *sock, struct sockaddr * uaddr, int addr_len, int flags)
{
        struct sock *sk=sock->sk;
        int err;
        long timeo;
        lock_sock(sk);
        if (uaddr->sa_family == AF_UNSPEC)
        {
            err = sk->prot->disconnect(sk, flags);
            sock->state = err ? SS_DISCONNECTING : SS_UNCONNECTED;
            goto out;
        }
        switch (sock->state)
        {
            default: err = -EINVAL;
                     goto out;
            case SS_CONNECTED: err = -EISCONN;
                     goto out;
            …….. }}
```

Connectionless socket

inet_stream_connect (2)

```
int inet_stream_connect(struct socket *sock, struct sockaddr * uaddr, int addr_len, int flags)
{
        ……..
        switch (sock->state)
        {
        ……..
        case SS_CONNECTING: err = -EALREADY; /* Fall out of switch with err, set for this state */
                break;
        case SS_UNCONNECTED: err = -EISCONN;
                if (sk->state != TCP_CLOSE) goto out;
                …….
                err = sk->prot->connect(sk, uaddr, addr_len);
                if (err < 0) goto out;
                sock->state = SS_CONNECTING;
                err = -EINPROGRESS;
                break;
        }
}
```

Call function:tcp_v4_connect()->tcp_connect()
->tcp_set_state(TCP_SYN_SENT) : send SYN packet

inet_stream_connect(3)

```
int inet_stream_connect(struct socket *sock, struct sockaddr * uaddr, int addr_len, int flags)
{
        ……..
        timeo = sock_sndtimeo(sk, flags&O_NONBLOCK);
        if ((1<<sk->state)&(TCPF_SYN_SENT|TCPF_SYN_RECV))
        {
           if (!timeo || !inet_wait_for_connect(sk, timeo)) goto out;
           err = sock_intr_errno(timeo);
           if (signal_pending(current)) goto out;
        }
        sock->state = SS_CONNECTED;
        err = 0;
out:
        release_sock(sk);
        return err;
}
```

Socket accept

sys_accept

```
asmlinkage long sys_accept(int fd, struct sockaddr *upeer_sockaddr, int *upeer_addrlen)
{
        struct socket *sock, *newsock;
        int err, len;
        char address[MAX_SOCK_ADDR];
        sock = sockfd_lookup(fd, &err);
        if (!sock) goto out;
        err = -EMFILE;
        if (!(newsock = sock_alloc())) goto out_put;
        newsock->type = sock->type;
        newsock->ops = sock->ops;
        err = sock->ops->accept(sock, newsock, sock->file->f_flags);
        if (err < 0) goto out_release;
        ……..
        if ((err = sock_map_fd(newsock)) < 0) goto out_release;
out_put:
        sockfd_put(sock);
out: return err;
out_release: sock_release(newsock);
        goto out_put;
}
```

Allocate a new socket to handle every incoming connection

Call function: inet_accept()

Return a new file descriptor

inet_accept

```
int inet_accept(struct socket *sock, struct socket *newsock, int flags)
{
        struct sock *sk1 = sock->sk;
        struct sock *sk2;
        int err = -EINVAL;
        if((sk2 = sk1->prot->accept(sk1,flags,&err)) == NULL) goto do_err;

        lock_sock(sk2);

        BUG_TRAP((1<<sk2->state)&(TCPF_ESTABLISHED|TCPF_CLOSE_WAIT|TCPF_CLOSE));

        sock_graft(sk2, newsock);

        newsock->state = SS_CONNECTED;
        release_sock(sk2);
        return 0;
do_err:
        return err;
}
```

Call function: tcp_accept(), set new sock to TCP_SYN_RECV state

sk2 should be in TCP_SYN_RECV state

newsock->sk = sk2;
sk2->socket = newsock;

Socket Send

send

- send(), sendto(), and sendmsg() are used to transmit a message to another socket.
- send() may be used only when the socket is in a connected state, while sendto() and sendmsg() may be used at any time.
- SYS_SEND, SYS_SENDTO, SYS_SENDMSG use different parameters
- sock_sendmsg()

sys_sendto

```
asmlinkage long sys_sendto(int fd, void * buff, size_t len, unsigned flags, struct sockaddr *addr, int addr_len)
{
        struct socket *sock;
        char address[MAX_SOCK_ADDR];
        int err;
        struct msghdr msg;
        struct iovec iov;
        sock = sockfd_lookup(fd, &err);
        msg.msg_name=NULL;
        msg.msg_iov=&iov;
        ……..
        if (sock->file->f_flags & O_NONBLOCK)
           flags |= MSG_DONTWAIT;
        msg.msg_flags = flags;
        err = sock_sendmsg(sock, &msg, len);
out_put:
        sockfd_put(sock);
out: return err;
 }
```

```
int sock_sendmsg(struct socket *sock, struct msghdr *msg,
int size)
{
   int err;
   struct scm_cookie scm;
   err = scm_send(sock, msg, &scm);
   if (err >= 0) {
      err = sock->ops->sendmsg(sock, msg, size, &scm);
      scm_destroy(&scm);
   }
   return err;
}
```

inet_sendmsg

```
int inet_sendmsg(struct socket *sock, struct msghdr *msg, int size, struct scm_cookie *scm)
{
      struct sock *sk = sock->sk;
      /* We may need to bind the socket. */
      if (sk->num==0 && inet_autobind(sk) != 0)
          return -EAGAIN;
      return sk->prot->sendmsg(sk, msg, size);
}
```

Call function tcp_sendmsg

Socket Receive

recv

- recv(), recvfrom() and recvmsg() are used to received data from a socket.
- The recvfrom() and recvmsg() may be used to receive data on a socket whether or not it is connection-oriented.
- sock_recvmsg()

sock_recvmsg

```
int sock_recvmsg(struct socket *sock, struct msghdr *msg, int size, int flags)
{
    struct scm_cookie scm;
    memset(&scm, 0, sizeof(scm));
    size = sock->ops->recvmsg(sock, msg, size, flags, &scm);
    if (size >= 0)
        scm_recv(sock, msg, &scm, flags);
    return size;
}
```

Call function: inet_recvmsg()

inet_recvmsg

```
int inet_recvmsg(struct socket *sock, struct msghdr *msg, int size,int flags, struct scm_cookie
      *scm)
{
      struct sock *sk = sock->sk;
      int addr_len = 0;
      int err;
      err = sk->prot->recvmsg(sk, msg, size, flags&MSG_DONTWAIT,flags&~MSG_DONTWAIT,
      &addr_len);
      if (err >= 0)
         msg->msg_namelen = addr_len;
      return err;
}
```

Call function: tcp_recvmsg() =>tcp_prequeue_process(sk)
⇒sk->backlog_rcv() == tcp_v4_rcv() => tcp_v4_do_rcv()
⇒tcp_rcv_state_process()

tcp_rcv_state_process(1)

```
int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb, struct tcphdr *th, unsigned len)
{
        ……..
        switch (sk->state) {
           case TCP_CLOSE: goto discard;
           case TCP_LISTEN:
                   if (th->ack) return 1;
                   if (th->syn) {
                       if (tp->af_specific->conn_request(sk, skb) < 0) return 1;
                       goto discard;
                   }
                   goto discard;
           case TCP_SYN_SENT:
                   queued = tcp_rcv_synsent_state_process(sk,skb,th,len);
                   ……
        }
        ……..
}
```

received ACK

received SYN

If SYN_ACK is received , function tcp_ack() is called and state is set to TCP_ESTABLISHED

tcp_rcv_state_process(2)

```
int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb, struct tcphdr *th, unsigned len)
{
        ……..
        if (th->ack)
        {
            int acceptable = tcp_ack(sk, skb, FLAG_SLOWPATH);
            switch(sk->state) {
                case TCP_SYN_RECV:
                    if (acceptable) {
                        tcp_set_state(sk, TCP_ESTABLISHED);
                        ……..
                    }
                case TCP_FIN_WAIT1:
                    ………
                    tcp_set_state(sk,TCP_FIN_WAIT2);
                    ……..
        }
        ……..
}
```

tcp_rcv_state_process(3)

```
int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb, struct tcphdr *th, unsigned len)
{
        ……..
        if (th->ack)
        {
           ……..
           switch(sk->state) {
              case TCP_CLOSING:
                   ........
                   tcp_time_wait(sk, TCP_TIME_WAIT, 0);
                   ........
              case TCP_LAST_ACK:
                   ……..
                   tcp_done(sk);
                   ……..
           }
        else
           goto discard;
```

enter TCP_TIME_WAIT state

state is set to TCP_CLOSE